

Table of Contents

- 1. Maintain prodockit 2
 - 1.1 The maintenance cycle 2
 - 1.2 Choose the right maintenance tool 3
 - 1.3 What automation does—and does not—prove 4
 - 1.4 Suggested cadence 4

1. Maintain prodockit

This section is for maintainers of the prodockit repository. It covers the package's own source, test matrix, documentation artifacts, pinned build inputs, GitHub Actions workflows, PyPI release, and public documentation deployment.

For a documentation project rather than this package repository, start with the default read-only [diagnostics guide](#). It maps every `pdk diag` result to an author remediation without requiring package-maintainer knowledge.

If you are writing and publishing a document with prodockit, use [Publish a document](#) instead. Machine setup, `prodockit-template`, template syncing, PDF generation, and Pages publication belong to that author workflow.

1.1 The maintenance cycle

The maintenance cycle below keeps repository identity, dependency versions, generated artifacts, and release automation under review.

1. Start from a clean default branch

Update the branch GitHub publishes and confirm that no local work will be mixed into the maintenance change:

```
git switch main
git pull --ff-only
git status --short
```

Create a branch for the work rather than maintaining directly on `main`:

```
git switch -c maintenance/update-build-inputs
```

2. Check repository identity

Confirm that the website links, edit buttons, brand icon, and README badges still describe the remote this checkout uses:

```
prodockit sync-repo --check
```

If it reports drift, run `prodockit sync-repo`, review the diff, and repeat the check. See [Repository metadata](#).

3. Check pinned build inputs

First check that every file agrees without asking the network for newer releases:

```
prodockit pins --check --offline
```

Use the scheduled drift workflow to decide whether to adopt a newer release. Do not upgrade merely to clear a notification: rebuild and compare the output first. See [Version pinning and drift](#).

4. Build the deliverables locally

Build in the same order as GitHub Actions. The PDF consumes the completed Zensical site, so the website comes first:

```
zensical build --clean --strict
prodockit pdf
```

Then run the checks appropriate to the repository. For prodockit itself:

```
ruff check .
mypy src
prodockit pins --check --offline
pytest
```

5. Review, push, and use the pull request gates

Review both source and generated output. Commit the source changes, push the branch, and open a pull request:

```
git diff --check
git status --short
git add <reviewed-files>
git commit -m "Maintain project build inputs"
git push -u origin HEAD
gh pr create
```

GitHub Actions runs the test matrix and strict documentation build on the pull request. Merge only after those required checks pass. Package releases have additional gates described in [Build and release](#).

1.2 Choose the right maintenance tool

Match the maintenance need to its safest starting tool in [Table 14.1](#).

Table 1.1. Choose the right maintenance tool

Need	Start with	What changes
A project, installation, or renderer fails unexpectedly	<code>pdk diag</code> , then <code>pdk diag --dry-run</code>	Nothing until interactive <code>--apply</code> ; every eligible action has a separate default-No decision and confirmation
A fork, mirror, or renamed repository points at the wrong place	<code>prodockit sync-repo --check</code>	Repository URLs, edit links, host icon, and managed README badges

Need	Start with	What changes
CI files disagree about dependency versions	<code>prodockit pins --check --offline</code>	Version declarations, preserving each file's existing operator
A newer dependency may change published output	GitHub's <code>drift.yml</code> result	Nothing automatically; it opens or updates an issue with the comparison
A prodockit release is ready	The release checklist	Package version, release notes, GitHub release, PyPI package, and rebuilt documentation

1.3 What automation does—and does not—prove

The workflows deliberately divide responsibility:

[Table 14.2](#) separates the checks performed by automation from the decisions that still need a person.

Table 1.2. What automation does—and does not—prove

Workflow	Trigger	Answer
<code>ci.yml</code>	Pull requests and pushes to <code>main</code>	Does the code work across supported Python versions, and do the docs build strictly?
<code>docs.yml</code>	Pushes to <code>main</code> and manual dispatch	Can the complete website and PDF be built, tested, deployed, and verified live?
<code>drift.yml</code>	Weekly schedule and manual dispatch	Would newer rendering dependencies change the published artifacts?
<code>publish.yml</code>	Published GitHub release	Can the tagged source build and publish to PyPI through Trusted Publishing?
<code>release-redeploy.yml</code>	Published GitHub release	Can <code>docs.yml</code> be rerun against <code>main</code> so the site sees the new tag?

A green workflow proves the question in its own row in [Table 14.2](#), not every row. In particular, a passing pull request does not publish PyPI, and a successful Pages deployment does not by itself prove the public URL serves the new bytes. The deployment workflow performs that final delivery check separately.

1.4 Suggested cadence

[Table 14.3](#) turns the maintenance tools into a practical event-based routine.

Table 1.3. Suggested cadence

When	Maintenance
Every pull request	Repository and pin consistency, tests, lint, typing, strict docs build
Weekly	Let <code>drift.yml</code> compare pinned and newest renderers; triage the issue it opens
After moving or forking a repository	Run <code>sync-repo</code> , rebuild, and inspect canonical/edit links and badges
Before a package release	Complete the local build gates, merge the release PR, publish a GitHub release, verify PyPI and Pages

The [command-line map](#) inventories the public CLI that a maintainer must keep consistent. The pages after it explain each repository maintenance job in depth.